

C和C++实务精选 惊鸿一瞥

Addison Wesley

C陷阱与缺陷

C Traps and Pitfalls

falls

Andrew Koenig 著

高巍 译

王昕 审校

[修订说明]

仅仅为了掌上书苑制作！

第一次修订。改正了文中的大部分错别字和格式错误，并对一些句子依照中文的习惯进行了改写。

[译序]

那些自认为已经“学完”C语言的人，请你们仔细读阅读这篇文章吧。路还长，很多东西要学。我也是……

[概述]

C语言像一把雕刻刀，锋利，并且在技师手中非常有用。和任何锋利的工具一样，C会伤到那些不能掌握它的人。本文介绍C语言伤害粗心的人的方法，以及如何避免伤害。

[内容]

1 词法缺陷

编译器的第一个部分常被称为词法分析器（lexical　analyzer）。词法分析器检查组成程序的字符序列，并将它们划分为记号（token）一个记号是一个由一个或多个字符构成的序列，它在语言被编译时具有一个（相关地）统一的意义。在C中，　例如，记号->的意义和组成它的每个独立的字符具有明显的区别，而且其意义独立于->出现的上下文环境。

另外一个例子，考虑下面的语句：

```
if(x > big) big = x;
```

该语句中的每一个分离的字符都被划分为一个记号，除了关键字if和标识符big的两个实例。

事实上，C程序被两次划分为记号。首先是预处理器读取程序。它必须对程序进行记号划分以发现标识宏的标识符。它必须通过对每个宏进行求值来替换宏调用。最后，经过宏替换的程序又被汇集成字符流送给编译器。编译器再第二次将这个流划分为记号。

在这一节中，我们将探索对记号的意义的普遍的误解以及记号和组成它们的字符之间的关系。稍后我们将谈到预处理器。

1.1 = 不是 ==

从Algol派生出来的语言，如Pascal和Ada，用:=表示赋值而用=表示比较。而C语言则是用=表示赋值而用==表示比较。这是因为赋值的频率要高于比较，因此为其分配更短的符号。

此外，C还将赋值视为一个运算符，因此可以很容易地写出多重赋值（如a = b = c），并且可以将赋值嵌入到一个大的表达式中。

这种便捷导致了一个潜在的问题：可能将需要比较的地方写成赋值。因此，下面的语句好像看起来是要检查x是否等于y：

```
if(x = y)

foo();
```

而实际上是将x设置为y的值并检查结果是否非零。再考虑下面的一个希望跳过空格、制表符和换行符的循环：

```
while(c == ' ' || c = '\t' || c == '\n')

c = getc(f);
```

在与'\t'进行比较的地方程序员错误地使用=代替了==。这个“比较”实际上是将'\t'赋给c，然后判断c的（新的）值是否为零。因为'\t'不为零，这个“比较”将一直为真，因此这个循环会吃尽整个文件。这之后会发生什么取决于特定的实现是否允许一个程序读取超过文件尾部的部分。如果允许，这个循环会一直运行。

一些C编译器会对形如e1 = e2的条件给出一个警告以提醒用户。当你确实需要先对一个变量进行赋值之后再检查变量是否非零时，为了在这种编译器中避免警告信息，应考虑显式给出比较符。换句话说，将：

```
if(x = y)

foo();
```

改写为：

```
if((x = y) != 0)

foo();
```

这样可以清晰地表示你的意图。

1.2 & 和 | 不是 && 和 ||

容易将==错写为=是因为很多其他语言使用=表示比较运算。 其他容易写错的运算符还有&和&&，以及|和||，这主要是因为C语言中的&和|运算符于其他语言中具有类似功能的运算符大为不同。我们将在第4节中贴近地观察这些运算符。

1.3 多字符记号

一些C记号，如/、*和=只有一个字符。而其他一些C记号，如/*和==，以及标识符，具有多个字符。当C编译器遇到紧连在一起的/和*时，它必须能够决定是将这两个字符识别为两个分离的记号还是一个单独的记号。C语言参考手册说明了如何决定："如果输入流到一个给定的字符串为止已经被识别为记号，则应该包含下一个字符以组成能够构成记号的最长的字符串"（[译注]即通常所说的"最长子串原则"）。因此，如果/是一个记号的第一个字符，并且/后面紧随了一个*，则这两个字符构成了注释的开始，不管其他上下文环境。

下面的语句看起来像是将y的值设置为x的值除以p所指向的值：

```
y = x/*p /* p 指向除数 */;
```

实际上，/*开始了一个注释，因此编译器简单地吞噬程序文本，直到*/的出现。换句话说，这条语句仅仅把y的值设置为x的值，而根本没有看到p。将这条语句重写为：

```
y = x / *p /* p 指向除数 */;
```

或者干脆是

```
y = x / (*p) /* p指向除数 */;
```

它就可以做注释所暗示的除法了。

这种模棱两可的写法在其他环境中就会引起麻烦。例如，老版本的C使用=+表示现在版本中的+=。这样的编译器会将

```
a=-1;
```

视为

```
a =- 1;
```

或

```
a = a - 1;
```

这会让打算写

a = -1;

的程序员感到吃惊。 另一方面，这种老版本的C编译器会将

a=/*b;

断句为

a =/ *b;

尽管/*看起来像一个注释。

1.4 例外

组合赋值运算符如+=实际上是两个记号。因此，

a + /* strange */ = 1

和

a += 1

是一个意思。看起来像一个单独的记号而实际上是多个记号的只有这一个特例。特别地，

p - > a

是不合法的。它和

p -> a

不是同义词。

另一方面，有些老式编译器还是将=+视为一个单独的记号并且和+=是同义词。

1.5 字符串和字符

单引号和双引号在C中的意义完全不同，在一些混乱的上下文中它们会导致奇怪的结果而不是错误消息。

包围在单引号中的一个字符只是编写整数的另一种方法。这个整数是给定的字符在实现的对照序列中的一个对应的值。因此，在一个ASCII实现中，'a'和0141或97表示完全相同的东西。而一个包围在双引号中的字符串，只是编写一个有双引号之间的字符和一个附加的二进制值为零的字符所初始化的一个无名数组的指针的一种简短方法。

下面的两个程序片断是等价的：

```
printf("Hello world\n");
```

```
char hello[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\n', 0 };

printf(hello);
```

使用一个指针来代替一个整数通常会得到一个警告消息（反之亦然），使用双引号来代替单引号也会得到一个警告消息（反之亦然）。但对于不检查参数类型的编译器却除外。因此，用

```
printf('\n');
```

来代替

```
printf("\n");
```

通常会在运行时得到奇怪的结果。（[译注]提示：正如上面所说，'\n'表示一个整数，它被转换为了一个指针，这个指针所指向的内容是没有意义的。）

由于一个整数通常足够大，以至于能够放下多个字符，一些C编译器允许在一个字符常量中存放多个字符。这意味着用'yes'代替"yes"将不会被发现。后者意味着“分别包含y、e、s和一个空字符的四个连续存储器区域中的第一个的地址”，而前者意味着“在一些实现定义的样式中表示由字符y、e、s联合构成的一个整数”。这两者之间的任何一致性都纯属巧合。

2 句法缺陷

要理解C语言程序，仅了解构成它的记号是不够的。还要理解这些记号是如何构成声明、表达式、语句和程序的。尽管这些构成通常都是定义良好的，但这些定义有时候是有悖于直觉的或混乱的。

在这一节中，我们将着眼于一些不明显句法构造。

2.1 理解声明

我曾经和一些人聊过天，他们那时正在在编写在一个小型的微处理器上单机运行的C程序。当这台机器的开关打开的时候，硬件会调用地址为0处的子程序。

为了模仿电源打开的情形，我们要设计一条C语句来显式地调用这个子程序。经过一些思考，我们写出了下面的语句：

```
(*(void(*)())0)();
```

这样的表达式会令C程序员心惊胆战。但是，并不需要这样，因为他们可以在一个简单的规则的帮助下很容易地构造它：以你使用的方式声明它。

每个C变量声明都具有两个部分：一个类型和一组具有特定格式的、期望用来对该类型求值的表达式。最简单的表达式就是一个变量：

```
float f, g;
```

说明表达式f和g——在求值的时候——具有类型float。由于待求值的是表达式，因此可以自由地使用圆括号：

```
float ((f));
```

这表示((f))求值为float并且因此，通过推断，f也是一个float。

同样的逻辑用在函数和指针类型。例如：

```
float ff();
```

表示表达式ff()是一个float，因此ff是一个返回一个float的函数。类似地，

float *pf;

表示*pf是一个float并且因此pf是一个指向一个float的指针。

这些形式的组合声明对表达式是一样的。因此，

float *g(), (*h)();

表示*g()和(*h)()都是float表达式。由于()比*绑定得更紧密，*g()和*(g())表示同样的东西：g是一个返回指float指针的函数，而h是一个指向返回float的函数的指针。

当我们知道如何声明一个给定类型的变量以后，就能够很容易地写出一个类型的模型（cast）：只要删除变量名和分号并将所有的东西包围在一对圆括号中即可。因此，由于

float *g();

声明g是一个返回float指针的函数，所以(float *())就是它的模型。

有了这些知识的武装，我们现在可以准备解决(*(void(*)())0)()了。 我们可以将它分为两个部分进行分析。首先，假设我们有一个变量fp，它包含了一个函数指针，并且我们希望调用fp所指向的函数。可以这样写：

(*fp)();

如果fp是一个指向函数的指针，则*fp就是函数本身，因此(*fp)()是调用它的一种方法。(*fp)中的括号是必须的，否则这个表达式将会被分析为*(fp())。我们现在要找一个适当的表达式来替换fp。

这个问题就是我们的第二步分析。如果C可以读入并理解类型，我们可以写：

(*0)();

但这样并不行，因为*运算符要求必须有一个指针作为它的操作数。另

外，这个操作数必须是一个指向函数的指针，以保证*的结果可以被调用。因此，我们需要将0转换为一个可以描述“指向一个返回void的函数的指针”的类型。

如果fp是一个指向返回void的函数的指针，则(*fp)()是一个void值，并且它的声明将会是这样的：

```
void (*fp)();
```

因此，我们需要写：

```
void (*fp)();

(*fp)();
```

来声明一个哑变量。一旦我们知道了如何声明该变量，我们也就知道了如何将一个常数转换为该类型：只要从变量的声明中去掉名字即可。因此，我们像下面这样将0转换为一个“指向返回void的函数的指针”：

```
(void(*)())0
```

接下来，我们用(void(*)())0来替换fp：

```
(*(void(*)())0)();
```

结尾处的分号用于将这个表达式转换为一个语句。

在这里，我们解决这个问题时没有使用typedef声明。通过使用它，我们可以更清晰地解决这个问题：

```
typedef void (*funcptr)();

(*(funcptr)0)();
```

2.2 运算符并不总是具有你所想象的优先级

假设有一个声明了的常量FLAG，它是一个整数，其二进制表示中的某一位被置位（换句话说，它是2的某次幂），并且你希望测试一个整型变量flags该位是否被置位。通常的写法是：

```
if(flags & FLAG) ...
```

其意义对于很多C程序员都是很明确的：if语句测试括号中的表达式求值的结果是否为0。出于清晰的目的我们可以将它写得更明确：

```
if(flags & FLAG != 0) ...
```

这个语句现在更容易理解了。但它仍然是错的，因为!=比&绑定得更紧密，因此它被分析为：

```
if(flags & (FLAG != 0)) ...
```

这（偶尔）是可以的，如FLAG是1或0（！）的时候，但对于其他2的幂是不行的[2]。

假设你有两个整型变量，h和l，它们的值在0和15（含0和15）之间，并且你希望将r设置为8位值，其低位为l，高位为h。一种自然的写法是：

```
r = h << 4 + 1;
```

不幸的是，这是错误的。加法比移位绑定得更紧密，因此这个例子等价于：

```
r = h << (4 + l);
```

正确的方法有两种：

```
r = (h << 4) + l;
```

```
r = h << 4 | l;
```

避免这种问题的一个方法是将所有的东西都用括号括起来，但表达式中的括号过度就会难以理解，因此最好还是是记住C中的优先级。

不幸的是，这有15个，太困难了。然而，通过将它们分组可以变得容易。

绑定得最紧密的运算符并不是真正的运算符：下标、函数调用和结构选择。这些都与左边相关联。

接下来是一元运算符。它们具有真正的运算符中的最高优先级。由于函数调用比一元运算符绑定得更紧密，你必须写(*p)()来调用p指向的函数；*p()表示p是一个返回一个指针的函数。转换是一元运算符，并且和其他一元运算符具有相同的优先级。一元运算符是右结合的，因此*p++表示*(p++)，而不是(*p)++。

在接下来是真正的二元运算符。其中数学运算符具有最高的优先级，然后是移位运算符、关系运算符、逻辑运算符、赋值运算符，最后是条件运算符。需要记住的两个重要的东西是：

所有的逻辑运算符具有比所有关系运算符都低的优先级。

移位运算符比关系运算符绑定得更紧密，但又不如数学运算符。

在这些运算符类别中，有一些奇怪的地方。乘法、除法和求余具有相同的优先级，加法和减法具有相同的优先级，以及移位运算符具有相同的优先级。

还有就是六个关系运算符并不具有相同的优先级：==和!=的优先级比其他关系运算符要低。这就允许我们判断a和b是否具有与c和d相同的顺序，例如：

a < b == c < d

在逻辑运算符中，没有任何两个具有相同的优先级。按位运算符比所有顺序运算符绑定得都紧密，每种与运算符都比相应的或运算符绑定得更紧密，并且按位异或（^）运算符介于按位与和按位或之间。

三元运算符的优先级比我们提到过的所有运算符的优先级都低。这可以保证选择表达式中包含的关系运算符的逻辑组合特性，如：

z = a < b && b < c ? d : e

这个例子还说明了赋值运算符具有比条件运算符更低的优先级是有意义的。另外，所有的复合赋值运算符具有相同的优先级并且是自右至左结合的，因此

a = b = c

和

```
b = c; a = b;
```

是等价的。

具有最低优先级的是逗号运算符。这很容易理解，因为逗号通常在需要表达式而不是语句的时候用来替代分号。

赋值是另一种运算符，通常具有混合的优先级。例如，考虑下面这个用于复制文件的循环：

```
while(c = getc(in) != EOF)

putc(c, out);
```

这个while循环中的表达式看起来像是c被赋以getc(in)的值，接下来判断是否等于EOF以结束循环。不幸的是，赋值的优先级比任何比较操作都低，因此c的值将会是getc(in)和EOF比较的结果，并且会被抛弃。因此，“复制”得到的文件将是一个由值为1的字节流组成的文件。

上面这个例子正确的写法并不难：

```
while((c = getc(in)) != EOF)

putc(c, out);
```

然而，这种错误在很多复杂的表达式中却很难被发现。例如，随UNIX系统一同发布的lint程序通常带有下面的错误行：

```
if (((t = BTYPE(pt1->aty) == STRTY) || t == UNIONTY) {
```

这条语句希望给t赋一个值，然后看t是否与STRTY或UNIONTY相等。而实际的效果却大不相同[3]。

C中的逻辑运算符的优先级具有历史原因。B语言——C的前辈——具有和C中的&和|运算符对应的逻辑运算符。尽管它们的定义是按位的 ，但编译器在条件判断上下文中将它们视为和&&和||一样。当在C中将它们分开后，优先级的改变是很危险的[4]。

2.3 看看这些分号！

C中的一个多余的分号通常会带来一点点不同：或者是一个空语句，无任何效果；或者编译器可能提出一个诊断消息，可以方便除去掉它。一个重要的区别是在必须跟有一个语句的if和while语句中。考虑下面的例子：

if(x > *big);*

big = x;

这不会发生编译错误，但这段程序的意义与：

if(x > big)

big = x;

就大不相同了。第一个程序段等价于：

if(x > big) { }

big = x;

也就是等价于：

big = x;

（除非*x*、*i*或*big*是带有副作用的宏）。

另一个因分号引起巨大不同的地方是函数定义前面的结构声明的末尾（*[*译注*]*这句话不太好听，看例子就明白了）。考虑下面的程序片段：

struct foo {

int x;

}

f() {

...

}

在紧挨着*f*的第一个*}*后面丢失了一个分号。它的效果是声明了一个函数*f*，返回值类型是*struct foo*，这个结构成了函数声明的一部分。如果这里出现了分号，则*f*将被定义为具有默认的整型返回值*[5]*。

2.4 switch语句

通常C中的switch语句中的case段可以进入下一个。例如，考虑下面的C和Pascal程序片断：

```
switch(color) {

case 1: printf ("red");

break;

case 2: printf ("yellow");

break;

case 3: printf ("blue");

break;

}

case color of

1: write ('red');

2: write ('yellow');

3: write ('blue');

end
```

这两个程序片段都作相同的事情：根据变量color的值是1、2还是3打印red、yellow或blue（没有新行符）。这两个程序片段非常相似，只有一点不同：Pascal程序中没有C中相应的break语句。C中的case标签是真正的标签：控制流程可以无限制地进入到一个case标签中。

看看另一种形式，假设C程序段看起来更像Pascal：

```
switch(color) {

case 1: printf ("red");

case 2: printf ("yellow");

case 3: printf ("blue");

}
```

并且假设color的值是2。则该程序将打印yellowblue，因为控制自然地转入到下一个printf()的调用。

这既是C语言switch语句的优点又是它的弱点。说它是弱点，是因为很容易忘记一个break语句，从而导致程序出现隐晦的异常行为。说它是优点，是因为通过故意去掉break语句，可以很容易实现其他方法难以实现的控制结构。尤其是在一个大型的switch语句中，我们经常发现对一个case的处理可以简化其他一些特殊的处理。

例如，设想有一个程序是一台假想的机器的翻译器。这样的一个程序可能包含一个switch语句来处理各种操作码。在这样一台机器上，通常减法在对其第二个运算数进行变号后就变成和加法一样了。因此，最好可以写出这样的语句：

```
case SUBTRACT:

opnd2 = -opnd2;

/* no break; */

case ADD:
```

```
...
```

另外一个例子，考虑编译器通过跳过空白字符来查找一个记号。这里，我们将空格、制表符和新行符视为是相同的，除了新行符还要引起行计数器的增长外：

```
case '\n':

linecount++;

/* no break */

case '\t':

case ' ':

...
```

2.5 函数调用

和其他程序设计语言不同，C要求一个函数调用必须有一个参数列表，但可以没有参数。因此，如果f是一个函数，

```
f();
```

就是对该函数进行调用的语句，而

```
f;
```

什么也不做。它会作为函数地址被求值，但不会调用它[6]。

2.6 悬挂**else**问题

在讨论任何语法缺陷时我们都不会忘记提到这个问题。尽管这一问题不是C语言所独有的，但它仍然伤害着那些有着多年经验的C程序员。

考虑下面的程序片断：

```
if(x == 0)
```

```
if(y == 0) error();

else {

z = x + y;

f(&z);

}
```

写这段程序的程序员的目的明显是将情况分为两种：x = 0和x != 0。在第一种情况中，程序段什么都不做，除非y = 0时调用error()。第二种情况中，程序设置z = x + y并以z的地址作为参数调用f()。

然而， 这段程序的实际效果却大为不同。其原因是一个else总是与其最近的if相关联。如果我们希望这段程序能够按照实际的情况运行，应该这样写：

```
if(x == 0) {

if(y == 0)

error();

else {

z = x + y;

f(&z);

}

}
```

换句话说，当x != 0发生时什么也不做。如果要达到第一个例子的效果，应该写：

```
if(x == 0) {
```

```
if(y ==0)

error();

}

else {

z = z + y;

f(&z);

}
```

3 连接

一个C程序可能有很多部分组成，它们被分别编译，并由一个通常称为连接器、连接编辑器或加载器的程序绑定到一起。由于编译器一次通常只能看到一个文件，因此它无法检测到需要程序的多个源文件的内容才能发现的错误。

在这一节中，我们将看到一些这种类型的错误。有一些C实现，但不是所有的，带有一个称为lint的程序来捕获这些错误。如果具有一个这样的程序，那么无论怎样地强调它的重要性都不过分。

3.1 你必须自己检查外部类型

假设你有一个C程序，被划分为两个文件。其中一个包含如下声明：

```
int n;
```

而令一个包含如下声明：

```
long n;
```

这不是一个有效的C程序，因为一些外部名称在两个文件中被声明为不同的类型。然而，很多实现检测不到这个错误，因为编译器在编译其中一个文件时并不知道另一个文件的内容。因此，检查类型的工作只能由连接器（或一些工具程序如lint）来完成；如果操作系统的连接器不能

识别数据类型，C编译器也没法过多地强制它。

那么，这个程序运行时实际会发生什么？这有很多可能性：

实现足够聪明，能够检测到类型冲突。则我们会得到一个诊断消息，说明n在两个文件中具有不同的类型。

你所使用的实现将int和long视为相同的类型。典型的情况是机器可以自然地进行32位运算。在这种情况下你的程序或许能够工作，好象你两次都将变量声明为long（或int）。但这种程序的工作纯属偶然。

n的两个实例需要不同的存储，它们以某种方式共享存储区，即对其中一个的赋值对另一个也有效。这可能发生，例如，编译器可以将int安排在long的低位。不论这是基于系统的还是基于机器的，这种程序的运行同样是偶然。

n的两个实例以另一种方式共享存储区，即对其中一个赋值的效果是对另一个赋以不同的值。在这种情况下，程序可能失败。

这种情况发生的里一个例子出奇地频繁。程序的某一个文件包含下面的声明：

```
char filename[] = "etc/passwd";
```

而另一个文件包含这样的声明：

```
char *filename;
```

尽管在某些环境中数组和指针的行为非常相似，但它们是不同的。在第一个声明中，filename是一个字符数组的名字。尽管使用数组的名字可以产生数组第一个元素的指针，但这个指针只有在需要的时候才产生并且不会持续。在第二个声明中，filename是一个指针的名字。这个指针可以指向程序员让它指向的任何地方。如果程序员没有给它赋一个值，它将具有一个默认的0值（NULL）（[译注]实际上，在C中一个为初始化的指针通常具有一个随机的值，这是很危险的！）。

这两个声明以不同的方式使用存储区，它们不可能共存。

避免这种类型冲突的一个方法是使用像lint这样的工具（如果可以的

话）。为了在一个程序的不同编译单元之间检查类型冲突，一些程序需要一次看到其所有部分。典型的编译器无法完成，但lint可以。

避免该问题的另一种方法是将外部声明放到包含文件中。这时，一个外部对象的类型仅出现一次[7]。

4 语义缺陷

一个句子可以是精确拼写的并且没有语法错误，但仍然没有意义。在这一节中，我们将会看到一些程序的写法会使得它们看起来是一个意思，但实际上是另一种完全不同的意思。

我们还要讨论一些表面上看起来合理但实际上会产生未定义结果的环境。我们这里讨论的东西并不保证能够在所有的C实现中工作。我们暂且忘记这些能够在一些实现中工作但可能不能在另一些实现中工作的东西，直到第7节讨论可以执行问题为止。

4.1 表达式求值顺序

一些C运算符以一种已知的、特定的顺序对其操作数进行求值。但另一些不能。例如，考虑下面的表达式：

```
a < b && c < d
```

C语言定义规定a < b首先被求值。如果a确实小于b，c < d必须紧接着被求值以计算整个表达式的值。但如果a大于或等于b，则c < d根本不会被求值。

要对a < b求值，编译器对a和b的求值就会有一个先后。但在一些机器上，它们也许是并行进行的。

C中只有四个运算符&&、||、?:和,指定了求值顺序。&&和||最先对左边的操作数进行求值，而右边的操作数只有在需要的时候才进行求值。而?:运算符中的三个操作数：a、b和c，最先对a进行求值，之后仅对b或c中的一个进行求值，这取决于a的值。,运算符首先对左边的操作数进行求值，然后抛弃它的值，对右边的操作数进行求值[8]。

C中所有其它的运算符对操作数的求值顺序都是未定义的。事实上，赋

值运算符不对求值顺序做出任何保证。

出于这个原因，下面这种将数组x中的前n个元素复制到数组y中的方法是不可行的：

```
i = 0;

while(i < n)

y = x[i++];
```

其中的问题是*y*的地址并不保证在*i*增长之前被求值。在某些实现中，这是可能的；但在另一些实现中却不可能。另一种情况出于同样的原因会失败：

```
i = 0;

while(i < n)

y[i++] = x;
```

而下面的代码是可以工作的：

```
i = 0;

while(i < n) {

y = x;

i++;

}
```

当然，这可以简写为：

```
for(i = 0; i < n; i++)

y = x;
```

4.2 &&、||和!运算符

C中有两种逻辑运算符，在某些情况下是可以交换的：按位运算符&、|和~，以及逻辑运算符&&、||和!。一个程序员如果用某一类运算符替换相应的另一类运算符会得到某些奇怪的效果：程序可能会正确地工作，但这纯属偶然。

&&、||和!运算符将它们的参数视为仅有“真”或“假”，通常约定0代表“假”而其它的任意值都代表“真”。这些运算符返回1表示“真”而返回0表示“假”，而且&&和||运算符当可以通过左边的操作数确定其返回值时，就不会对右边的操作数进行求值。

因此!10是零，因为10非零；10 && 12是1，因为10和12都非零；10 || 12也是1，因为10非零。另外，最后一个表达式中的12不会被求值，10 || f()中的f()也不会被求值。

考虑下面这段用于在一个表中查找一个特定元素的程序：

```
i = 0;

while(i < tabsize && tab != x)

i++;
```

这段循环背后的意思是如果*i*等于*tabsize*时循环结束，元素未被找到。否则，*i*包含了元素的索引。

假设这个例子中的*&&*不小心被替换为了*&*，这个循环可能仍然能够工作，但只有两种幸运的情况可以使它停下来。

首先，这两个操作都是当条件为假时返回*0*，当条件为真时返回*1*。只要*x*和*y*都是*1*或*0*，*x & y*和*x && y*都具有相同的值。然而，如果当使用了除*1*之外的非零值表示“真”时互换了这两个运算符，这个循环将不会工作。

其次，由于数组元素不会改变，因此越过数组最后一个元素前进一个位置时是无害的，循环会幸运地停下来。失误的程序会越过数组的结尾，因为*&*不像*&&*，总是会对所有的操作数进行求值。因此循环的最后一次获取*tab*时*i*的值已经等于*tabsize*了。如果*tabsize*是*tab*中元素的数量，

则会取到*tab*中不存在的一个值。

4.3 下标从零开始

在很多语言中，具有n个元素的数组其元素的号码和它的下标是从1到n严格对应的。但在C中不是这样。

一个具有n个元素的C数组中没有下标为n的元素，其中的元素的下标是从0到n - 1。因此从其它语言转到C语言的程序员应该特别小心地使用数组：

int i, a[10];

for(i = 1; i <= 10; i++)

a = *0;*

这个例子的目的是要将*a*中的每个元素都设置为*0*，但没有期望的效果。因为*for*语句中的比较*i < 10*被替换成了*i <= 10*，*a*中的一个编号为*10*的并不存在的元素被设置为了*0*，这样内存中*a*后面的一个字被破坏了。如果编译该程序的编译器按照降序地址为用户变量分配内存，则*a*后面就是*i*。将*i*设置为零会导致该循环陷入一个无限循环。

4.4 C并不总是转换实参

下面的程序段由于两个原因会失败：

double s;

s = sqrt(2);

printf("%g\n", s);

第一个原因是sqrt()需要一个double值作为它的参数，但没有得到。第二个原因是它返回一个double值但没有这样声名。改正的方法只有一个：

double s, sqrt();

s = sqrt(2.0);

```
printf("%g\n", s);
```

C中有两个简单的规则控制着函数参数的转换：(1)比int短的整型被转换为int；(2)比double短的浮点类型被转换为double。所有的其它值不被转换。确保函数参数类型的正确性是程序员的责任。

因此，一个程序员如果想使用如sqrt()这样接受一个double类型参数的函数，就必须仅传递给它float或double类型的参数。常数2是一个int，因此其类型是错误的。

当一个函数的值被用在表达式中时，其值会被自动地转换为适当的类型。然而，为了完成这个自动转换，编译器必须知道该函数实际返回的类型。没有更进一步声名的函数被假设返回int，因此声名这样的函数并不是必须的。然而，sqrt()返回double，因此在成功使用它之前必须要声名。

实际上，C实现通常允许一个文件包含include语句来包含如sqrt()这些库函数的声名，但是对那些自己写函数的程序员来说，编写声名也是必要的——或者说，对那些编写非凡的C程序的人来说是有必要的。

这里有一个更加壮观的例子：

```
main() {

int i;

char c;

for(i = 0; i < 5; i++) {

scanf("%d", &c);

printf("%d", i);

}

printf("\n");

}
```

表面上看，这个程序从标准输入中读取五个整数并向标准输出写入0 1 2 3 4。实际上，它并不总是这么做。譬如在一些编译器中，它的输出为0 0 0 0 0 1 2 3 4。

为什么？因为c的声名是char而不是int。当你令scanf()去读取一个整数时，它需要一个指向一个整数的指针。但这里它得到的是一个字符的指针。但scanf()并不知道它没有得到它所需要的：它将输入看作是一个指向整数的指针并将一个整数存贮到那里。由于整数占用比字符更多的内存，这样做会影响到c附近的内存。

c附近确切是什么是编译器的事；在这种情况下这有可能是i的低位。因此，每当向c中读入一个值，i就被置零。当程序最后到达文件结尾时，scanf()不再尝试向c中放入新值，i才可以正常地增长，直到循环结束。

4.5 指针不是数组

C程序通常将一个字符串转换为一个以空字符结尾的字符数组。假设我们有两个这样的字符串s和t，并且我们想要将它们连接为一个单独的字符串r。我们通常使用库函数strcpy()和strcat()来完成。下面这种明显的方法并不会工作：

```
char *r;

strcpy(r, s);

strcat(r, t);
```

这是因为r没有被初始化为指向任何地方。尽管r可能潜在地表示某一块内存，但这并不存在，直到你分配它。

让我们再试试，为r分配一些内存：

```
char r[100];

strcpy(r, s);

strcat(r, t);
```

这只有在s和t所指向的字符串不很大的时候才能够工作。不幸的是，C

要求我们为数组指定的大小是一个常数，因此无法确定r是否足够大。然而，很多C实现带有一个叫做malloc()的库函数，它接受一个数字并分配这么多的内存。通常还有一个函数称为strlen()，可以告诉我们一个字符串中有多少个字符：因此，我们可以写：

```
char *r, *malloc();

r = malloc(strlen(s) + strlen(t));

strcpy(r, s);

strcat(r, t);
```

然而这个例子会因为两个原因而失败。首先，malloc()可能会耗尽内存，而这个事件仅通过静静地返回一个空指针来表示。

其次，更重要的是，malloc()并没有分配足够的内存。一个字符串是以一个空字符结束的。而strlen()函数返回其字符串参数中所包含字符的数量，但不包括结尾的空字符。因此，如果strlen(s)是n，则s需要n + 1个字符来盛放它。因此我们需要为r分配额外的一个字符。再加上检查malloc()是否成功，我们得到：

```
char *r, *malloc();

r = malloc(strlen(s) + strlen(t) + 1);

if(!r) {

complain();

exit(1);

}

strcpy(r, s);

strcat(r, t);
```

4.6 避免提喻法

提喻法（Synecdoche,　sin-ECK-duh-key）是一种文学手法，有点类似于明喻或暗喻，在牛津英文词典中解释如下：“a more comprehensive term is used for a less comprehensive or vice versa; as whole for part or part for whole, genus for species or species for genus, etc.（将全面的单位用作不全面的单位，或反之；如整体对局部或局部对整体、一般对特殊或特殊对一般，等等。）”

这可以精确地描述C中通常将指针误以为是其指向的数据的错误。正将常会在字符串中发生。例如：

```
char *p, *q;

p = "xyz";
```

尽管认为p的值是xyz有时是有用的，但这并不是真的，理解这一点非常重要。p的值是指向一个有四个字符的数组中第0个元素的指针，这四个字符是'x'、'y'、'z'和'\0'。因此，如果我们现在执行：

```
q = p;
```

p和q会指向同一块内存。内存中的字符没有因为赋值而被复制。这种情况看起来是这样的：

要记住的是，复制一个指针并不能复制它所指向的东西。

因此，如果之后我们执行：

```
q[1] = 'Y';
```

q所指向的内存包含字符串xYz。p也是，因为p和q指向相同的内存。

4.7 空指针不是空字符串

将一个整数转换为一个指针的结果是实现相关的（implementation-dependent），除了一个例外。这个例外是常数0，它可以保证被转换为一个与其它任何有效指针都不相等的指针。这个值通常类似这样定义：

```
#define NULL 0
```

但其效果是相同的。要记住的一个重要的事情是，当用0作为指针时它决不能被解除引用。换句话说，当你将0赋给一个指针变量后，你就不能访问它所指向的内存。不能这样写：

```
if(p == (char *)0) ...
```

也不能这样写：

```
if(strcmp(p, (char *)0) == 0) ...
```

因为strcmp()总是通过其参数来查看内存地址的。

如果p是一个空指针，这样写也是无效的：

```
printf(p);
```

或

```
printf("%s", p);
```

4.8 整数溢出

C语言关于整数操作的上溢或下溢定义得非常明确。

只要有一个操作数是无符号的，结果就是无符号的，并且以2n为模，其中n为字长。如果两个操作数都是带符号的，则结果是未定义的。

例如，假设a和b是两个非负整型变量，你希望测试a + b是否溢出。一个明显的办法是这样的：

```
if(a + b < 0)

complain();
```

通常，这是不会工作的。

一旦a + b发生了溢出，对于结果的任何赌注都是没有意义的。例如，在某些机器上，一个加法运算会将一个内部寄存器设置为四种状态：正、负、零或溢出。　在这样的机器上，编译器有权将上面的例子实现为首

先将a和b加在一起，然后检查内部寄存器状态是否为负。如果该运算溢出，内部寄存器将处于溢出状态，这个测试会失败。

使这个特殊的测试能够成功的一个正确的方法是依赖于无符号算术的良好定义，即要在有符号和无符号之间进行转换：

```
if((int)((unsigned)a + (unsigned)b) < 0)

complain();
```

4.9 移位运算符

两个原因会令使用移位运算符的人感到烦恼：

在右移运算中，空出的位是用0填充还是用符号位填充？

移位的数量允许使用哪些数？

第一个问题的答案很简单，但有时是实现相关的。如果要进行移位的操作数是无符号的，会移入0。如果操作数是带符号的，则实现有权决定是移入0还是移入符号位。如果在一个右移操作中你很关心空位，那么用unsigned来声明变量。这样你就有权假设空位被设置为0。

第二个问题的答案同样简单：如果待移位的数长度为n，则移位的数量必须大于等于0并且严格地小于n。因此，在一次单独的操作中不可能将所有的位从变量中移出。

例如，如果一个int是32位，且n是一个int，写n << 31和n << 0是合法的，但n << 32和n << -1是不合法的。

注意，即使实现将符号为移入空位，对一个带符号整数的右移运算和除以2的某次幂也不是等价的。为了证明这一点，考虑(-1) >> 1的值，这是不可能为0的。[译注：(-1) / 2的结果是0。]

5 库函数

每个有用的C程序都会用到库函数，因为没有办法把输入和输出内建到语言中去。在这一节中，我们将会看到一些广泛使用的库函数在某种情况下会出现的一些非预期行为。

5.1 getc()返回整数

考虑下面的程序：

```
#include

main() {

char c;

while((c = getchar()) != EOF)

putchar(c);

}
```

这段程序看起来好像要将标准输入复制到标准输出。实际上，它并不完全会做这些。

原因是c被声明为字符而不是整数。这意味着它将不能接收可能出现的所有字符包括EOF。

因此这里有两种可能性。有时一些合法的输入字符会导致c携带和EOF相同的值，有时又会使c无法存放EOF值。在前一种情况下，程序会在文件的中间停止复制。在后一种情况下，程序会陷入一个无限循环。

实际上，还存在着第三种可能：程序会偶然地正确工作。C语言参考手册严格地定义了表达式

```
((c = getchar()) != EOF)
```

的结果。其6.1节中声明：

当一个较长的整数被转换为一个较短的整数或一个char时，它会被截去左侧；超出的位被简单地丢弃。

7.14节声明：

存在着很多赋值运算符，它们都是从右至左结合的。它们都需要一个左

值作为左侧的操作数，而赋值表达式的类型就是其左侧的操作数的类型。其值就是已经赋过值的左操作数的值。

这两个条款的组合效果就是必须通过丢弃getchar()的结果的高位，将其截短为字符，之后这个被截短的值再与EOF进行比较。作为这个比较的一部分，c必须被扩展为一个整数，或者采取将左侧的位用0填充，或者适当地采取符号扩展。

然而，一些编译器并没有正确地实现这个表达式。它们确实将getchar()的值的低几位赋给c。但在c和EOF的比较中，它们却使用了getchar()的值！这样做的编译器会使这个事例程序看起来能够“正确地”工作。

5.2 缓冲输出和内存分配

当一个程序产生输出时，能够立即看到它有多重要？这取决于程序。

例如，终端上显示输出并要求人们坐在终端前面回答一个问题，人们能够看到输出以知道该输入什么就显得至关重要了。另一方面，如果输出到一个文件中，并最终被发送到一个行式打印机，只有所有的输出最终能够到达那里是重要的。

立即安排输出的显示通常比将其暂时保存在一大块一起输出要昂贵得多。因此，C实现通常允许程序员控制产生多少输出后在实际地写出它们。

这个控制通常约定为一个称为setbuf()的库函数。如果buf是一个具有适当大小的字符数组，则

```
setbuf(stdout, buf);
```

将告诉I/O库写入到stdout中的输出要以buf作为一个输出缓冲，并且等到buf满了或程序员直接调用fflush()再实际写出。缓冲区的合适的大小在中定义为BUFSIZ。

因此，下面的程序解释了通过使用setbuf()来讲标准输入复制到标准输出：

```
#include
```

```
main() {

int c;

char buf[BUFSIZ];

setbuf(stdout, buf);

while((c = getchar()) != EOF)

putchar(c);

}
```

不幸的是，这个程序是错误的，因为一个细微的原因。

要知道毛病出在哪，我们需要知道缓冲区最后一次刷新是在什么时候。答案；主程序完成之后，库将控制交回到操作系统之前所执行的清理的一部分。在这一时刻，缓冲区已经被释放了！

有两种方法可以避免这一问题。

首先，使用静态缓冲区，或者将其显式地声明为静态：

```
static char buf[BUFSIZ];
```

或者将整个声明移到主函数之外。

另一种可能的方法是动态地分配缓冲区并且从不释放它：

```
char *malloc();

setbuf(stdout, malloc(BUFSIZ));
```

注意在后一种情况中，不必检查malloc()的返回值，因为如果它失败了，会返回一个空指针。而setbuf()可以接受一个空指针作为其第二个参数，这将使得stdout变成非缓冲的。这会运行得很慢，但它是可以运行的。

6 预处理器

运行的程序并不是我们所写的程序：因为C预处理器首先对其进行了转换。出于两个主要原因（和很多次要原因），预处理器为我们提供了一些简化的途径。

首先，我们希望可以通过改变一个数字并重新编译程序来改变一个特殊量（如表的大小）的所有实例[9]。

其次，我们可能希望定义一些东西，它们看起来象函数但没有函数调用所需的运行开销。例如，putchar()和getchar()通常实现为宏以避免对每一个字符的输入输出都要进行函数调用。

6.1 宏不是函数

由于宏可以象函数那样出现，有些程序员有时就会将它们视为等价的。因此，看下面的定义：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

注意宏体中所有的括号。它们是为了防止出现a和b是带有比>优先级低的表达式的情况。

一个重要的问题是，像max()这样定义的宏每个操作数都会出现两次并且会被求值两次。因此，在这个例子中，如果a比b大，则a就会被求值两次：一次是在比较的时候，而另一次是在计算max()值的时候。

这不仅是低效的，还会发生错误：

```
biggest = x[0];

i = 1;

while(i < n)

biggest = max(biggest, x[i++]);
```

当max()是一个真正的函数时，这会正常地工作，但当max()是一个宏的时候会失败。譬如，假设x[0]是2、x[1]是3、x[2]是1。我们来看看在第

一次循环时会发生什么。赋值语句会被扩展为：

biggest = ((biggest) > (x[i++]) ? (biggest) : (x[i++]));

首先，biggest与x[i++]进行比较。由于i是1而x[1]是3，这个关系是"假"。其副作用是，i增长到2。

由于关系是"假"，x[i++]的值要赋给biggest。然而，这时的i变成2了，因此赋给biggest的值是x[2]的值，即1。

避免这些问题的方法是保证max()宏的参数没有副作用：

biggest = x[0];

for(i = 1; i < n; i++)

biggest = max(biggest, x*);*

还有一个危险的例子是混合宏及其副作用。这是来自*UNIX*第八版的中*putc()*宏的定义：

#define putc(x, p) (--(p)->_cnt >= 0 ? ((p)->_ptr++ = (x)) : _flsbuf(x, p))*

*putc()*的第一个参数是一个要写入到文件中的字符，第二个参数是一个指向一个表示文件的内部数据结构的指针。注意第一个参数完全可以使用如*z++之类的东西，尽管它在宏中两次出现，但只会被求值一次。而第二个参数会被求值两次（在宏体中，*x*出现了两次，但由于它的两次出现分别在一个:的两边，因此在*putc()*的一个实例中它们之中有且仅有一个被求值）。由于*putc()*中的文件参数可能带有副作用，这偶尔会出现问题。不过，用户手册文档中提到："由于*putc()*被实现为宏，其对待*stream*可能会具有副作用。特别是*putc(c, *f++)*不能正确地工作。"但是*putc(*c++, f)*在这个实现中是可以工作的。

有些*C*实现很不小心。例如，没有人能正确处理*putc(*c++, f)*。另一个例子，考虑很多*C*库中出现的*toupper()*函数。它将一个小写字母转换为相应的大写字母，而其它字符不变。如果我们假设所有的小写字母和所有的大写字母都是相邻的（大小写之间可能有所差距），我们可以得到这样的函数：

```
toupper(c) {

if(c >= 'a' && c <= 'z')

c += 'A' - 'a';

return c;

}
```

在很多*C*实现中，为了减少比实际计算还要多的调用开销，通常将其实现为宏：

```
#define toupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) + ('A' - 'a') : (c))
```

很多时候这确实比函数要快。然而，当你试着写*toupper(*p++)*时，会出现奇怪的结果。

另一个需要注意的地方是使用宏可能会产生巨大的表达式。例如，继续考虑*max()*的定义：

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

假设我们这个定义来查找*a*、*b*、*c*和*d*中的最大值。如果我们直接写：

```
max(a, max(b, max(c, d)))
```

它将被扩展为：

```
((a) > (((b) > (((c) > (d) ? (c) : (d))) ? (b) : (((c) > (d) ? (c) : (d))))) ?

(a) : (((b) > (((c) > (d) ? (c) : (d))) ? (b) : (((c) > (d) ? (c) : (d))))))
```

这出奇的庞大。我们可以通过平衡操作数来使它短一些：

```
max(max(a, b), max(c, d))
```

这会得到：

```
((((a) > (b) ? (a) : (b))) > (((c) > (d) ? (c) : (d))) ?

(((a) > (b) ? (a) : (b))) : (((c) > (d) ? (c) : (d))))
```

这看起来还是写：

```
biggest = a;

if(biggest < b) biggest = b;

if(biggest < c) biggest = c;

if(biggest < d) biggest = d;
```

比较好一些。

6.2 宏不是类型定义

宏的一个通常的用途是保证不同地方的多个事物具有相同的类型：

```
#define FOOTYPE struct foo

FOOTYPE a;

FOOTYPE b, c;
```

这允许程序员可以通过只改变程序中的一行就能改变a、b和c的类型，尽管a、b和c可能声明在很远的不同地方。

使用这样的宏定义还有着可移植性的优势——所有的C编译器都支持它。很多C编译器并不支持另一种方法：

```
typedef struct foo FOOTYPE;
```

这将FOOTYPE定义为一个与struct foo等价的新类型。

这两种为类型命名的方法可以是等价的，但typedef更灵活一些。例如，考虑下面的例子：

```
#define T1 struct foo *

typedef struct foo * T2;
```

这两个定义使得T1和T2都等价于一个struct foo的指针。但看看当我们试图在一行中声明多于一个变量的时候会发生什么：

```
T1 a, b;

T2 c, d;
```

第一个声明被扩展为：

```
struct foo * a, b;
```

这里a被定义为一个结构指针，但b被定义为一个结构（而不是指针）。相反，第二个声明中c和d都被定义为指向结构的指针，因为T2的行为好像真正的类型一样。

7 可移植性缺陷

C被很多人实现并运行在很多机器上。这也正是在一个地方写的C程序应该能够很容易地转移到另一个编程环境中去的原因。

然而，由于有很多的实现者，它们并不和其他人交流。此外，不同的系统有不同的需求，因此一台机器上的C实现和另一台上的多少会有些不同。

由于很多早期的C实现都关系到UNIX操作系统，因此这些函数的性质都是专于该系统的。当一些人开始在其他系统中实现C时，他们尝试使库的行为类似于UNIX系统中的行为。

但他们并不总是能够成功。更有甚者，很多人从UNIX系统的不同版本入手，一些库函数的本质不可避免地发生分歧。今天，一个C程序员如果想写出对于不同环境中的用户都有用的程序就必须知道很多这些细微的差别。

7.1 一个名字中都有什么？

一些C编译器将一个标识符中的所有字符视为签名。而另一些在存储标识符时会忽略一个极限之外的所有字符。C编译器产生的目标程序同将要被加载器进行处理以访问库中的子程序。加载器对于它们能够处理的名字通常应用自己的约束。

一个常见的加载器约束是所有的外部名字必须只能是大写的。面对这样的加载器约束，C实现者会强制要求所有的外部名字都是大写的。这种约束在C语言参考手册中第2.1节由所描述。

一个标识符是一个字符和数字序列，第一个字符必须是一个字母。下划线_算作字母。大写字母和小写字母是不同的。只有前八个字符是签名，但可以使用更多的字符。可以被多种汇编器和加载器使用的外部标识符，有着更多的限制：

这里，参考手册中继续给出了一些例子如有些实现要求外部标识符具有单独的大小写格式、或者少于八个字符、或者二者都有。

正因为所有这些，在一个希望可以移植的程序中小心地选择标识符是很重要的。为两个子程序选择print_fields和print_float这样的名字不是个好办法。

考虑下面这个显著的函数：

```
char *Malloc(unsigned n) {

char *p, *malloc();

p = malloc(n);

if(p == NULL)

panic("out of memory");

return p;

}
```

这个函数是保证耗尽内存而不会导致没有检测的一个简单的办法。程序员可以通过调用Mallo()来代替malloc()。如果malloc()不幸失败，将调用

panic()来显示一个恰当的错误消息并终止程序。

然而，考虑当该函数用于一个忽略大小写区别的系统中时会发生什么。这时，名字malloc和Malloc是等价的。换句话说，库函数malloc()被上面的Malloc()函数完全取代了，当调用malloc()时它调用的是它自己。显然，其结果就是第一次尝试分配内存就会陷入一个递归循环并随之发生混乱。但在一些能够区分大小写的实现中这个函数还是可以工作的。

7.2 一个整数有多大？

C为程序员提供三种整数尺寸：普通、短和长，还有字符，其行为像一个很小的整数。C语言定义对各种整数的大小不作任何保证：

整数的四种尺寸是非递减的。

普通整数的大小要足够存放任意的数组下标。

字符的大小应该体现特定硬件的本质。

许多现代机器具有8位字符，不过还有一些具有7位获9位字符。因此字符通常是7、8或9位。

长整数通常至少32位，因此一个长整数可以用于表示文件的大小。

普通整数通常至少16位，因为太小的整数会更多地限制一个数组的最大大小。

短整数总是恰好16位。

在实践中这些都意味着什么？最重要的一点就是别指望能够使用任何一个特定的精度。非正式情况下你可以假设一个短整数或一个普通整数是16位的，而一个长整数是32位的，但并不保证总是会有这些大小。你当然可以用普通整数来压缩表大小和下标，但当一个变量必须存放一个一千万的数字的时候呢？

一种更可移植的做法是定义一个“新的”类型：

```
typedef long tenmil;
```

现在你就可以使用这个类型来声明一个变量并知道它的宽度了，最坏的情况下，你也只要改变这个单独的类型定义就可以使所有这些变量具有正确的类型。

7.3 字符是带符号的还是无符号的？

很多现代计算机支持8位字符，因此很多现代C编译器将字符实现为8位整数。然而，并不是所有的编译器都按照同将的方式解释这些8位数。

这些问题在将一个char制转换为一个更大的整数时变得尤为重要。对于相反的转换，其结果却是定义良好的：多余的位被简单地丢弃掉。但一个编译器将一个char转换为一个int却需要作出选择：将char视为带符号量还是无符号量？如果是前者，将char扩展为int时要复制符号位；如果是后者，则要将多余的位用0填充。

这个决定的结果对于那些在处理字符时习惯将高位置1的人来说非常重要。这决定着8位的字符范围是从-128到127还是从0到255。这又影响着程序员对哈希表和转换表之类的东西的设计。

如果你关心一个字符值最高位置一时是否被视为一个负数，你应该显式地将它声明为unsigned char。这样就能保证在转换为整数时是基0的，而不像普通char变量那样在一些实现中是带符号的而在另一些实现中是无符号的。

另外，还有一种误解是认为当c是一个字符变量时，可以通过写(unsigned)c来得到与c等价的无符号整数。这是错误的，因为一个char值在进行任何操作（包括转换）之前转换为int。这时c会首先转换为一个带符号整数再转换为一个无符号整数，这会产生奇怪的结果。

正确的方法是写(unsigned char)c。

7.4 右移位是带符号的还是无符号的？

这里再一次重复：一个关心右移操作如何进行的程序最好将所有待移位的量声明为无符号的。

7.5 除法如何舍入？

假设我们用b除a得到商为q余数为r：

q = a / b;

r = a % b;

我们暂时假设b > 0。

我们期望a、b、q和r之间有什么关联？

最重要的，我们期望q * b + r == a，因为这是对余数的定义。

如果a的符号发生改变，我们期望q的符号也发生改变，但绝对值不变。

我们希望保证r >= 0且r < b。例如，如果余数将作为一个哈希表的索引，它必须要保证总是一个有效的索引。

这三点清楚地描述了整数除法和求余操作。不幸的是，它们不能同时为真。

考虑3 / 2，商1余0。这满足第一点。而-3 / 2的值呢？根据第二点，商应该是-1，但如果是这样的话，余数必须也是-1，这违反了第三点。或者，我们可以通过将余数标记为1来满足第三点，但这时根据第一点商应该是-2。这又违反了第二点。

因此C和其他任何实现了整数除法舍入的语言必须放弃上述三个原则中的至少一个。

很多程序设计语言放弃了第三点，要求余数的符号必须和被除数相同。这可以保证第一点和第二点。很多C实现也是这样做的。

然而，C语言的定义只保证了第一点和|r| < |b|以及当a >= 0且b > 0时r >= 0。 这比第二点或第三点的限制要小，实际上有些编译器满足第二点或第三点，但不太常见（如一个实现可能总是向着距离0最远的方向进行舍入）。

尽管有些时候不需要灵活性，C语言还是足够可以让我们令除法完成我们所要做的、提供我们所想知道的。例如，假设我们有一个数n表示一个标识符中的字符的一些函数，并且我们想通过除法得到一个哈希表入

口h，其中0 <= h <= HASHSIZE。如果我们知道n是非负的，我们可以简单地写：

```
h = n % HASHSIZE;
```

然而，如果n有可能是负的，这样写就不好了，因为h可能也是负的。然而，我们知道h > -HASHSIZE，因此我们可以写：

```
h = n % HASHSIZE;

if(n < 0)

h += HASHSIZE;
```

同样，将n声明为unsigned也可以。

7.6 一个随机数有多大？

这个尺寸是模糊的，还受库设计的影响。在PDP-11[10]机器上运行的仅有的C实现中，有一个称为rand()的函数可以返回一个（伪）随机非负整数。PDP-11中整数长度包括符号位是16位，因此rand()返回一个0到215-1之间的整数。

当C在VAX-11上实现时，整数的长度变为32位长。那么VAX-11上的rand()函数返回值范围是什么呢？

对于这个系统，加利福尼亚大学的人认为rand()的返回值应该涵盖所有可能的非负整数，因此它们的rand()版本返回一个0到231-1之间的整数。

而AT&T的人则觉得如果rand()函数仍然返回一个0到215之间的值　则可以很容易地将PDP-11中期望rand()能够返回一个小于215的值的程序移植到VAX-11上。

因此，现在还很难写出不依赖实现而调用rand()函数的程序。

7.7 大小写转换

toupper()和tolower()函数有着类似的历史。他们最初都被实现为宏：

```
#define toupper(c) ((c) + 'A' - 'a')

#define tolower(c) ((c) + 'A' - 'a')
```

当给定一个小写字母作为输入时，toupper()将产生相应的大写字母。tolower()反之。这两个宏都依赖于实现的字符集，它们需要所有的大写字母和对应的小写字母之间的差别都是常数的。这个假设对于ASCII和EBCDIC字符集来说都是有效的，可能不是很危险，因为这些不可移植的宏定义可以被封装到一个单独的文件中并包含它们。

这些宏确实有一个缺陷，即：当给定的东西不是一个恰当的字符，它会返回垃圾。因此，下面这个通过使用这些宏来将一个文件转为小写的程序是无法工作的：

```
int c;

while((c = getchar()) != EOF)

putchar(tolower(c));
```

我们必须写：

```
int c;

while((c = getchar()) != EOF)

putchar(isupper(c) ? tolower(c) : c);
```

就这一点，AT&T中的UNIX开发组织提醒我们，toupper()和tolower()都是事先经过一些适当的参数进行测试的。考虑这样重写这些宏：

```
#define toupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) + 'A' - 'a' : (c))

#define tolower(c) ((c) >= 'A' && (c) <= 'Z' ? (c) + 'a' - 'A' : (c))
```

但要知道，这里c的三次出现都要被求值，这会破坏如toupper(*p++)这样的表达式。因此，可以考虑将toupper()和tolower()重写为函数。toupper()看起来可能像这样：

```
int toupper(int c) {

if(c >= 'a' && c <= 'z')

return c + 'A' - 'a';

return c;

}
```

tolower()类似。

这个改变带来更多的问题，每次使用这些函数的时候都会引入函数调用开销。我们的英雄认为一些人可能不愿意支付这些开销，因此他们将这个宏重命名为：

```
#define _toupper(c) ((c) + 'A' - 'a')

#define _tolower(c) ((c) + 'a' - 'A')
```

这就允许用户选择方便或速度。

这里面其实只有一个问题：伯克利的人们和其他的C实现者并没有跟着这么做。　　　　　这意味着一个在AT&T系统上编写的使用了toupper()或tolower()的程序，如果没有为其传递正确大小写字母参数，在其他C实现中可能不会正常工作。

如果不知道这些历史，可能很难对这类错误进行跟踪。

7.8 先释放，再重新分配

很多C实现为用户提供了三个内存分配函数：malloc()、realloc()和free()。调用malloc(n)返回一个指向有n个字符的新分配的内存的指针，这个指针可以由程序员使用。给free()传递一个指向由malloc()分配的内存的指针可以使这块内存得以再次使用。通过一个指向已分配区域的指针和一个新的大小调用realloc()可以将这块内存扩大或缩小到新尺寸，这个过程中可能要复制内存。

也许有人会想，真相真是有点微妙啊。下面是System V接口定义中出现

的对realloc()的描述：

realloc改变一个由ptr指向的size个字节的块，并返回该块（可能被移动）的指针。　　在新旧尺寸中比较小的一个尺寸之下的内容不会被改变。

而UNIX系统第七版的参考手册中包含了这一段的副本。此外，还包含了描述realloc()的另外一段：

如果在最后一次调用malloc、realloc或calloc后释放了ptr所指向的块，realloc依旧可以工作；因此，free、malloc和realloc的顺序可以利用malloc压缩存贮的查找策略。

因此，下面的代码片段在UNIX第七版中是合法的：

```
free (p);

p = realloc(p, newsize);
```

这一特性保留在从UNIX第七版衍生出来的系统中：可以先释放一块存储区域，然后再重新分配它。这意味着，在这些系统中释放的内存中的内容在下一次内存分配之前可以保证不变。因此，在这些系统中，我们可以用下面这种奇特的思想来释放一个链表中的所有元素：

```
for(p = head; p != NULL; p = p->next)

free((char *)p);
```

而不用担心调用free()会导致p->next不可用。

不用说，这种技术是不推荐的，因为不是所有C实现都能在内存被释放后将它的内容保留足够长的时间。然而，第七版的手册遗留了一个未声明的问题：realloc()的原始实现实际上是必须要先释放再重新分配的。出于这个原因，一些C程序都是先释放内存再重新分配的，而当这些程序移植到其他实现中时就会出现问题。

7.9 可移植性问题的一个实例

让我们来看一个已经被很多人在很多时候解决了的问题。下面的程序带

有两个参数：一个长整数和一个函数（的指针）。它将整数转换位十进制数，并用代表其中每一个数字的字符来调用给定的函数。

```
void printnum(long n, void (*p)()) {

if(n < 0) {

(*p)('-');

n = -n;

}

if(n >= 10)

printnum(n / 10, p);

(*p)(n % 10 + '0');

}
```

这个程序非常简单。首先检查n是否为负数；如果是，则打印一个符号并将n变为正数。接下来，测试是否n >= 10。如果是，则它的十进制表示中包含两个或更多个数字，因此我们递归地调用printnum()来打印除最后一个数字外的所有数字。最后，我们打印最后一个数字。

这个程序——由于它的简单——具有很多可移植性问题。首先是将n的低位数字转换成字符形式的方法。用n % 10来获取低位数字的值是好的，但为它加上'0'来获得相应的字符表示就不好了。这个加法假设机器中顺序的数字所对应的字符数顺序的，没有间隔，因此'0' + 5和'5'的值是相同的，等等。尽管这个假设对于ASCII和EBCDIC字符集是成立的，但对于其他一些机器可能不成立。避免这个问题的方法是使用一个表：

```
void printnum(long n, void (*p)()) {

if(n < 0) {

(*p)('-');
```

```
n = -n;

}

if(n >= 10)

printnum(n / 10, p);

(*p)("0123456789"[n % 10]);

}
```

另一个问题发生在当n < 0时。这时程序会打印一个负号并将n设置为-n。这个赋值会发生溢出，因为在使用2的补码的机器上通常能够表示的负数比正数要多。例如，一个（长）整数有k位和一个附加位表示符号，则-2k可以表示而2k却不能。

解决这一问题有很多方法。最直观的一种是将n赋给一个unsigned long值。然而，一些C便一起可能没有实现unsigned long，因此我们来看看没有它怎么办。

在第一个实现和第二个实现的机器上，改变一个正整数的符号保证不会发生溢出。问题仅出在改变一个负数的符号时。因此，我们可以通过避免将n变为正数来避免这个问题。

当然，一旦我们打印了负数的符号，我们就能够将负数和正数视为是一样的。下面的方法就强制在打印符号之后n为负数，并且用负数值完成我们所有的算法。如果我们这么做，我们就必须保证程序中打印符号的部分只执行一次；一个简单的方法是将这个程序划分为两个函数：

```
void printnum(long n, void (*p)()) {

if(n < 0) {

(*p)('-');

printneg(n, p);

}
```

```
else

printneg(-n, p);

}

void printneg(long n, void (*p)()) {

if(n <= -10)

printneg(n / 10, p);

(*p)("0123456789"[-(n % 10)]);

}
```

printnum()现在只检查要打印的数是否为负数；如果是的话则打印一个符号。否则，它以n的负绝对值来调用printneg()。我们同时改变了printneg()的函数体来适应n永远是负数或零这一事实。

我们得到什么？我们使用n / 10和n % 10来获取n的前导数字和结尾数字（经过适当的符号变换）。调用整数除法的行为在其中一个操作数为负的时候是实现相关的。因此，n % 10有可能是正的！这时，-(n % 10)是负数，将会超出我们的数字字符数组的末尾。

为了解决这一问题，我们建立两个临时变量来存放商和余数。作完除法后，我们检查余数是否在正确的范围内，如果不是的话则调整这两个变量。printnum()没有改变，因此我们只列出printneg()：

```
void printneg(long n, void (*p)()) {

long q;

int r;

if(r > 0) {

r -= 10;
```

```
q++;
}
if(n <= -10) {
printneg(q, p);
}
(*p)("0123456789"[-r]);
}
```

8 这里是空闲空间

还有很多可能让C程序员误入迷途的地方本文没有提到。如果你发现了，请联系作者。在以后的版本中它会被包含进来，并添加一个表示感谢的脚注。

参考

《The C Programming Language》（Kernighan and Ritchie, Prentice-Hall 1978）是最具权威的C著作。它包含了一个优秀的教程，面向那些熟悉其他高级语言程序设计的人，和一个参考手册，简洁地描述了整个语言。尽管自1978年以来这门语言发生了不少变化，这本书对于很多主题来说仍然是个定论。这本书同时还包含了本文中多次提到的“C语言参考手册”。

《The C Puzzle Book》（Feuer, Prentice-Hall, 1982）是一本少见的磨炼人们文法能力的书。这本书收集了很多谜题（和答案），它们的解决方法能够测试读者对于C语言精妙之处的知识。

《C: A Referenct Manual》（Harbison and Steele, Prentice Hall 1984）是特意为实现者编写的一本参考资料。其他人也会发现它是特别有用的——因为他能从中参考细节。

脚注

1. 这本书是基于图书《C Traps and Pitfalls》（Addison-Wesley，1989，ISBN 0-201-17928-8）的一个扩充，有兴趣的读者可以读一读它。

2. 因为!=的结果不是1就是0。

3. 感谢Guy Harris为我指出这个问题。

4. Dennis Ritchie和Steve Johnson同时向我指出了这个问题。

5. 感谢一位不知名的志愿者提出这个问题。

6. 感谢Richard Stevens指出了这个问题。

7. 一些C编译器要求每个外部对象仅有一个定义，但可以有多个声明。使用这样的编译器时，我们何以很容易地将一个声明放到一个包含文件中，并将其定义放到其它地方。这意味着每个外部对象的类型将出现两次，但这比出现多于两次要好。

8. 分离函数参数用的逗号不是逗号运算符。例如在f(x，y)中，x和y的获取顺序是未定义的，但在g((x，y))中不是这样的。其中g只有一个参数。它的值是通过对x进行求值、抛弃这个值、再对y进行求值来确定的。

9. 预处理器还可以很容易地组织这样的显式常量以能够方便地找到它们。

10. PDP-11和VAX-11是数组设备集团（DEC）的商标。

Table of Contents